Tactics and Parameters

Gueorgui Jojgov [1]

Faculteit Wiskunde en Informatica
Technische Universiteit Eindhoven
The Netherlands

Abstract

In this paper we discuss the problem of internalizing the meta-level transformations between (representations of) incomplete proofs and terms in a theorem prover based on Type Theory. These transformations (usually referred to as *tactics*) can be seen as meta-level functions between terms representing the state of the theorem prover. Starting with parameterized variables as representations of unknown terms, we propose an extension of the Pure Type Systems (PTSs) with parameterized abstractions. We show that such a system can adequately represent *instances* of tactics, i.e. the mapping between a state and the state resulting from it by the application of a given tactic.

We establish the important meta-theoretical properties of the extended system such as confluence, subject reduction, normalization, etc.

1 Introduction

Ever since the ground-breaking work of de Bruijn on AUTOMATH [5] there has been intensive work on mechanical tools to formalize and check mathematical theories using Type Theory. In interactive theorem provers based on Type Theory one tries to construct interactively a term inhabiting a given type. Under the formulas-as-types interpretation, such a term would be an encoding of a proof of the proposition encoded by the type. Because of the undecidability of the inhabitation problem (for interesting enough systems) one has to construct the proof-terms interactively and is hence forced to work with partially constructed objects. This raises many questions about the representation and manipulation of incomplete objects in type theory and logic. Most of the research effort in the area of open terms has been dedicated to the representation problem while the formalization of manipulations of incomplete objects has stayed on the meta-level on the background.

[1] Email: G.I.Jojgov@tue.nl

The formalization of incomplete terms and proofs allows us to treat unknowns and incomplete terms containing unknowns as first-class objects and therefore to model inside a calculus the incomplete objects that one works with in a theorem prover. We are able to represent the *states* of a theorem prover by open terms. The manipulations of the incomplete objects however are done on the meta-level. This means that we have no formalization of the transition between the states of the prover. It is clear that to give full formalization of the process of interactive term construction one also needs to have representation of the transitions between the states. In many systems these transitions are called *tactics* and they may be of significant complexity. Some systems use full-blown programming languages (e.g. ML) as a tactic language. This is not surprising as tactics often involve pattern matching and/or unification, recursion, backtracking, complex decision and search procedures, failure handling etc. All this however happens at the meta-level outside the object calculus that we work with. In this paper we make a first step towards internalizing some meta-level transformations by providing a calculus that allows us to represent states as terms and transitions between them as functional terms. We do that by extending a calculus of open terms with abstractions over unknowns and function types over them. On a more technical side, this means that we start with a representation of unknowns by parameterized variables (e.g. $x[\Delta]$, where Δ is a list of variable declarations) in a Pure Type System (PTS) and allow abstractions over them. This lead us to a calculus with parameterized $\lambda-$abstractions (e.g. $\lambda x[\Delta]{:}A.M$) and $\Pi-$abstractions (e.g. $\Pi x[\Delta]{:}A.M$). The resulting calculus is not powerful enough to describe arbitrary tactics because it lack essential mechanisms for doing that (e.g. unification and recursion), but it is capable of describing *tactic instances*, i.e. mappings between individual states arising from a specific applications of tactics.

The paper is organized as follows: in Section 2 we give a brief description of the representation of unknowns by (hereditarily) parameterized variables and by means of examples we describe the problems and the solutions that we propose to address them. After discussing related work in Section 2.3, we introduce our extension of PTSs in Section 3 where we define the syntax, the reduction rules and the typing system. We establish some meta-properties of the system like confluence, subject reduction and normalization. We conclude with a discussion on future work in Section 4.

2 Motivation and Related Work

2.1 Representing Unknowns in Open Terms by Parameterized Variables

Throughout this paper we model unknowns that appear in terms by parameterized variables that we also call *meta-variables*. This is of course only one of the many possibilities that have been studied (see Section 2.3), but we choose this approach because it is well-suited for representing incomplete log-

ical derivations and terms (see [7,8]). In this section we will briefly point to the main issues concerning this representation and introduce notation that we will use later.

Meta-variables stem from the use of higher-order variables to represent unknown terms in unification algorithms by Miller [13]. Indeed, a parameterized variable can be seen as a higher-order function of its arguments. We need the arguments in order to record substitutions carried out in a term as a result of β-reduction. If we would like to model a function of an argument x of type A with unknown body, we can introduce a meta-variable $h[x{:}A]$ with a parameter of type A representing the unknown body and the function is then given by the term $\lambda x{:}A.h[x]$. We can apply this function to arguments $(\lambda x{:}A.h[x])b$ and even compute the result of the beta-reduction: $h[b]$. We see that the parameters help us 'remember' that the unknown represented by h was subject to a substitution. This is very important because we would like to be able to instantiate h at any time and always get the same result. So, if we instantiate $h[x{:}A]$ by x before the beta-reduction we get $(\lambda x{:}A.x)b$ that reduces to b and if we instantiate it in $h[b]$ we get b again. In other words, *parameters make instantiation and reduction commute.*

Commutation of instantiation and reduction is obtained also in the other systems of open terms known in the literature, but parameters have an advantage when one looks at the logical side of the problem. As discussed in Jojgov [8], in incomplete logical terms and proofs there are two kinds of abstractions – one is the object-level abstraction and the other one is the meta-level dependency of unknown objects on the variables that occur free in them. These dependencies need to be kept separate in order to get a faithful extension of the formulas-as-types embedding of logic in type theory to incomplete proofs and terms. To illustrate this, consider the following incomplete derivations and their translations to type theory judgments where meta-variables are represented by higher-order function variables:

$$\frac{?}{A \to B} \qquad \frac{\dfrac{\dfrac{[A]^i}{?}}{B}}{A \to B}{}^{i} \qquad \frac{[A]^i \quad \dfrac{?}{A \to B}}{\dfrac{B}{A \to B}{}^{i}}$$

$$(a) \qquad (b) \qquad (c)$$

$$f_a : A \to B \vdash f_a : A \to B$$
$$f_b : A \to B \vdash \lambda x{:}A.\,(f_b\, x) : A \to B$$
$$f_c : A \to B \vdash \lambda x{:}A.\,(f_c\, x) : A \to B$$

The '?'-symbols here represent missing part of the proof with conclusion the formula given below and assumptions given above the symbol. We notice several things: first, looking at the representation of the unknowns in the typing judgment we cannot distinguish between the three because they are all represented by a variable of type $A \to B$. This denies us the opportunity to track the progress being made towards solving the unknown. Second, we notice that

the derivations (b) and (c), although very different from a logical viewpoint (they have different sets of possible completions to finished derivations), have identical translations. We can track the problem down to the identification in the typing judgment of the object- and meta-level abstractions present in the logical system.

Parameters help us distinguish between the representations of the two levels of abstraction. The meta-dependencies are recorded as parameters. This approach has also the advantage that it avoids the need to extend the object-level function space to accommodate the meta-level dependencies. The above examples translated to a system where unknowns are represented by parameterized meta-variables look like this:

$$\begin{aligned} m_a[\,] : A \to B \vdash & \qquad m_a[\,] : A \to B \\ m_b[x : A] : B \quad \vdash & \quad \lambda p{:}A.m_b[p] : A \to B \\ m_c[\,] : A \to B \vdash \lambda x{:}A.(m_c[\,]\, x) & : A \to B \end{aligned}$$

A further discussion on the possible forms of incompleteness in logical proofs and terms yields terms containing bound variables whose object-level binders have not (yet) been constructed. Such terms occur naturally in the setting of forward proof constructions that correspond to the building of a proof term from the leaves to the root. We can view such incomplete terms as unknown terms that have known subterms. As unknowns are represented by meta-variables, the known subterms can be given as arguments to the meta-variables. To account for the binding of the variables in the subterms, we need to give meta-level binding power to the meta-variables. Then a typical meta-variable instance looks like this $m[\langle\Delta_1\rangle M_1 \ldots \langle\Delta_n\rangle M_n]$. Each M_i represents a known subterm and the variables declared in Δ_i are those that are supposed to be bound by the yet unconstructed binders. As discussed in [8], to achieve that we need to use *hereditarily parameterized meta-variables*, i.e. meta-variables whose parameters can be parameterized themselves. A logic-based argument similar to the examples above can be given (see [8]) as to why we need to use parameters instead of object-level abstractions that may even be unavailable in the system (e.g. higher-order functions in the framework of first-order logic).

2.2 Representing States and Tactic Instances as Terms

In the previous section we introduced the parameterized meta-variables as a mechanism to model incomplete terms. The process of stepwise construction of a (proof)term can be modelled by a sequence of open terms representing the incomplete proof at different stages. Let us take as an example the following problem: Assume that A is a type and a, b and c are terms of this type. Assume that R is a binary relation on A that is transitive and for each x $R(x, b)$ holds. As a part of a larger proof we would like to prove $R(a, c)$. We can reduce this goal to the goal of proving $R(b, c)$ using the assumptions we

have. The initial state of the prover can be depicted as:

```
thm  :  (x:A)(R x b)
 tr  :  (x,y,z:A)(R x y)->(R y z)->(R x z)
===========
?  :  (R a c)
```

The declarations above the line are the assumptions under which we have to prove the goal $R(a, c)$. Let us collect them in the context Δ:

$$\Delta = thm{:}\Pi x{:}A.Rxb, tr{:}\Pi x, y, z{:}A.Rxy \to Ryz \to Rxz$$

We can represent the unknown proof of the goal by a meta-variable m with parameters Δ and type Rac. Then the initial state of the prover can be encoded by the judgment

$$m[\Delta]{:}Rac \vdash \lambda\Delta.m[thm, tr] : Rac$$

where $\lambda\Delta.M$ of course means $\lambda thm{:}\ldots.\lambda tr{:}\ldots.M$. At this moment we would like to use the transitivity of R by instantiating x and z by a and c. This produces three new goals – to find an instantiation for y in the transitivity, and to prove the premises corresponding to Rxy and Ryz:

```
                thm  :  ...        thm  :  ...
thm  :  ...      tr  :  ...         tr  :  ...
 tr  :  ...      y?  :  A           y?  :  A
===========     ===========        ===========
y?  :  A        p?  :  (R a y?)    q?  :  (R y?  c)
```

How do we encode this new state and how is it related to the previous one? We introduce a new meta-variable for each new goal and in the place of $m[thm, tr]$ we have an application of tr:

$$\begin{aligned}
&y?[\Delta]{:}A,\\
&p?[\Delta] : (R\,a\,y?[thm, tr]),\\
&q?[thm, tr] : (R\,y?[thm, tr]\,c),\\
&\vdash \lambda\Delta.(tr\,a\,y?[thm, tr]\,c\,p?[thm, tr]\,q?[thm, tr]) : Rac
\end{aligned}$$

Now we would like to use our other assumption, thm, to solve the second goal. At this point a theorem prover would use unification to match $R\,a\,y?$ to $R\,x\,b$ and find out that in order to apply thm, x has to be instantiated by a and $y?$ has to be b. This results in the following state:

```
thm  :  ...
 tr  :  ...
===========
r?  :  (R b c)
```

And it can be represented by the judgment

$$r?[\Delta]{:}(R\,b\,c) \vdash \lambda\Delta.(tr\,a\,b\,c\,(thm\,a)\,r?[thm, tr]) : R\,a\,c$$

This does not complete the proof, but if we have a look at the two transitions between the three states, we notice that there are several steps that we do on the meta-level that are not part of our representation. We introduce new meta-variables, we use them to give solutions to (some of) the pre-existing ones and we propagate these solutions through the representation of the state. All these are the meta-steps we make at each of the two transitions. The question arises:

Can we make these meta-transformations explicit by internalizing them in the calculus?

In this paper we will give affirmative answer to this question by extending our system with abstractions over meta-variables as means to internalize the dependency of the state on its meta-variables. The corresponding application operation would play the role of explicit representation of the instantiation of meta-variables. In this system a state can be encapsulated in a term by abstracting out all its meta-variables. The transformation steps then become functions that expect terms of appropriate types matching the types of the state terms and can also be encoded by λ-terms.

As an illustration, using the abstractions and applications that we will introduce in Section 3, the first state can be encoded by $\lambda m[\Delta]{:}Rac.\lambda\Delta.m[thm, tr]$ and its type is $\Pi m[\Delta]{:}Rac.\Pi\Delta.Rac$. The transformation step leading to the second state can be given by:

$$\lambda S : (\Pi m[\Delta]{:}Rac.\Pi\Delta.Rac).$$
$$\lambda y?[\Delta]{:}A.$$
$$\lambda p?[\Delta] : (R\, a\, y?[thm, tr]).$$
$$\lambda q?[\Delta] : (R\, y?[thm, tr]\, c).$$
$$(S \cdot \langle\Delta\rangle(tr\, a\, y?[thm, tr]\, c\, p?[thm, tr], q?[thm, tr]))$$

If we apply this transformation term to the state term and normalize, we get the term

$$\lambda y?[\Delta]{:}A.$$
$$\lambda p?[\Delta] : (R\, a\, y?[thm, tr]).$$
$$\lambda q?[\Delta] : (R\, y?[thm, tr]\, c).$$
$$\lambda\Delta.(tr\, a\, y?[thm, tr]\, c\, p?[thm, tr]\, q?[thm, tr])$$

which is exactly the encoding of the second state.

2.3 Related Work

The work in this paper builds on several ideas already present in the field of open terms. The representation of holes by higher-order functions used

in Miller's work [13] on unification is in the basis of our approach to the unknowns, but we use it in a modified form because of the need to separate object- and meta-level level abstractions (see [8] for a discussion on this). This idea has been employed previously in Luo's PAL$^+$ logical framework [10] to avoid extending the object-level function space in order to accommodate meta-level functions. The handling of the scopes in open terms can be done in different ways, ALF [11] and Munoz [14] employ explicit substitutions similarly to Strecker's Typelab [16]. The use of parameters makes explicit the idea that is implicit in Typelab's handling of meta-variables where Strecker has noticed that it suffices to use explicit substitutions attached to meta-variables only. The idea to represent states as terms is introduced in the thesis of McBride [12] where he presents the OLEG framework of open terms. He uses binders for meta-variables to represent states, but it differs from our approach in several aspects. First, the meta-variables that we consider are parameterized. Instead, in OLEG the position of the binder is used to specify the context in which the meta-variable should be solved. Second, in our system the binders for meta-variables occurring in a term have a corresponding binder in the type. This means that the type of a term with a meta-variable binder may depend on the meta-variable. This allows us to make functions that expect terms with meta-variables as arguments. The corresponding application operation allows such a function to explicitly instantiate meta-variables in its arguments.

Another system that is closely related to our presentation is the $\lambda[\,]$-cube of Bognar [4]. In that system we have separate binders both for object-level and meta-level binders, and both of them have a corresponding binder on the type level. Our system differs from the $\lambda[\,]$-cube in that it allows hereditarily parameterized variables to be constructed and abstracted over, while in the cube the parameters cannot be parameterized themselves. In that sense our system extends the systems of the $\lambda[\,]$-cube.

The present work is a continuation of the previous discussions of open terms and proofs in higher-order logic by Geuvers and Jojgov [7,8] where the problem of extending the formulas-as-types embedding to incomplete proofs and terms has been discussed. There we internalized the notion of unknown in the calculus, in this paper we extend this formalization to the transformations of open terms.

We make heavy use of the parameter mechanism of Bloo, Kamareddine, Laan and Nederpelt [3] who extend earlier work of Poll and Severi [15]. We extend that work by introducing the more general notion of hereditary parametrization. The author believes that an extension of the C^{p}D^{p}PTSs described in that work to PTS with hereditarily parametrization and parameterized variables that results in PTS with hereditarily parameterized variables, constants and definitions (V^{h}C^{h}D^{h}PTSs) could be useful for modelling of many practical applications. Such an extension would be forthcoming in the author's thesis.

3 Pure Type Systems with Hereditarily Parameterized Variables

In this section we will present an extension of the Pure Type Systems (PTSs) introduced by Berardi [2] and Terlouw [18] as a generalization of the systems of Barendregt's λ-cube (see [1]). We assume that the reader is acquainted with the background facts about PTSs (see for example [1,6]).

We extend the standard definition of a PTS by adding parametrization to the λ- and Π-abstractions. A parameterized λ-abstraction $\lambda m[\Delta]{:}A.M$ represents a term that has abstracted out the meta-variable $m[\Delta]$ that potentially occurs in M. Such a term would have a parameterized Π-abstraction as a type: $\Pi m[\Delta]{:}A.B$. As the use of λ suggests, we can apply parameterized λ-abstractions to arguments and that would act as an explicit notation for the instantiation operation. Meta-variables however have parameters that can be used in the term that instantiates them. Therefore, we need to introduce the parameters of a meta-variable into the argument of an application: $(\lambda m[x{:}A]{:}A.m[z])\cdot\langle x{:}A\rangle x$. This term represents *explicitly* the instantiation of the meta-variable $m[x{:}A]$ by x in the term $m[z]$ (indeed, we will see that it β-reduces to z as expected). Notice how the extended application $M\cdot\langle x{:}A\rangle N$ introduces x in scope for the term N.

3.1 Syntax

Every PTS is given by a tuple $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}\rangle$ where the elements of $\mathcal{S}$ are called *sorts*, $\mathcal{A} \subseteq \mathcal{S}\times\mathcal{S}$ is the set of axioms and $\mathcal{R} \subseteq \mathcal{S}\times\mathcal{S}\times\mathcal{S}$ is a set of triples that restrict the formation of Π-types (see e.g.[6]). The set of the *pseudo-terms* of the extended PTS is given by the following grammar:

$$\mathcal{T} ::= \mathcal{S} \mid x[\langle\Delta\rangle\mathcal{T}\ldots\langle\Delta\rangle\mathcal{T}] \mid \mathcal{T}\cdot\langle\Delta\rangle\mathcal{T} \mid \lambda x[\Delta]{:}\mathcal{T}.\mathcal{T} \mid \Pi x[\Delta]{:}\mathcal{T}.\mathcal{T}$$
$$\Delta ::= \varepsilon \mid \Delta, x[\Delta]{:}\mathcal{T}$$

This definition is motivated by the intuitive meaning of the parameterized abstractions and application introduced above. $\lambda x[\Delta]{:}A.M$ and $\Pi x[\Delta]{:}A.M$ introduce the parameterized variable $x[\Delta]$ in M where it can be used provided it is given appropriate arguments. The variables declared in Δ can be used in A, but their scope does not extend to M. The term N is in the scope of the variables in Δ in an application $M\cdot\langle\Delta\rangle N$. Similarly, in a variable instance $x[\langle\Delta_1\rangle N_1\ldots\langle\Delta_n\rangle N_n]$, each N_i is in the scope of the variables in Δ_i.

In order to ease the notation, we identify the unparameterized variables and the variables with empty parameter lists.

On the level of contexts we define the notion of structural equivalence $\approx$ as follows:

$$\frac{}{\varepsilon \approx \varepsilon} \qquad \frac{\Gamma_1 \approx \Gamma_2 \quad \Delta_1 \approx \Delta_2}{\Gamma_1, x_1[\Delta_1]{:}A_1 \approx \Gamma_2, x_2[\Delta_2]{:}A_2}$$

The relation $\Delta \approx \Theta$ should be read as "Δ and Θ have the same structure". Note that the relation states properties of the structure of the contexts only. In particular, in the definition above A_1 and A_2 are not subject to any restrictions. We will assume that the names of the parameterized variables determine up to $\approx$-equivalence the context describing their parameters. Hence, if we talk about a variable $x[\Delta]$ then its instances $x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]$ must have the same number of actual parameters as there are elements in Δ and if Δ is the context $x_1[\Delta_1]{:}A_1, \ldots, x_n[\Delta_n]{:}A_n$ then $\Delta_i \approx \Theta_i$.

Note also that the structural equivalence relation is a weaker notion than α-equivalence as contexts that are not α-convertible can have the same structure. The need to introduce this notion arises from the possibility to do β-reductions in contexts that are explicitly recorded in terms (see the definition of β-reduction).

Example 3.1 [Well-formed pseudo-terms]

- If $x[y[z[p{:}D]{:}E]{:}F]$ is a parameterized variable then the following term is well-formed:

$$\lambda Y{:}(\Pi z[p{:}D]{:}A.B).x[\langle z[p{:}D]{:}E\rangle(Y \cdot \langle p{:}D\rangle z[p])]$$

- If $h[p[i{:}A]{:}B, q[j{:}A]{:}B \to C]{:}A \to C$ is a parameterized variable then $h[\langle i{:}A\rangle(a \cdot i), \langle j{:}A\rangle(b \cdot j)]$ is a well-formed term.

Definition 3.2 [Free and bound variables] The set of the free variables $\mathsf{FV}(-)$ of a term or a context is defined as follows:

$$\begin{aligned}
\mathsf{FV}(\varepsilon) &= \emptyset \\
\mathsf{FV}(\Delta, x[\Delta']{:}A) &= \mathsf{FV}(\Delta) \cup \mathsf{FV}(A) \setminus \mathsf{dom}(\Delta', \Delta) \cup \mathsf{FV}(\Delta') \setminus \mathsf{dom}(\Delta) \\
\mathsf{FV}(s) &= \emptyset \\
\mathsf{FV}(x[\langle\Delta_1\rangle M_1, \ldots, \langle\Delta_n\rangle M_n]) &= \{x\} \cup \bigcup\nolimits_j (\mathsf{FV}(M_j) \setminus \mathsf{dom}(\Delta_i) \cup \mathsf{FV}(\Delta_i)) \\
\mathsf{FV}(M \cdot \langle\Delta\rangle N) &= \mathsf{FV}(M) \cup \mathsf{FV}(N) \setminus \mathsf{dom}(\Delta) \cup \mathsf{FV}(\Delta) \\
\mathsf{FV}(\lambda x[\Delta]{:}A.M) &= \mathsf{FV}(M) \setminus \{x\} \cup \mathsf{FV}(A) \setminus \mathsf{dom}(\Delta) \cup \mathsf{FV}(\Delta) \\
\mathsf{FV}(\Pi x[\Delta]{:}A.M) &= \mathsf{FV}(M) \setminus \{x\} \cup \mathsf{FV}(A) \setminus \mathsf{dom}(\Delta) \cup \mathsf{FV}(\Delta)
\end{aligned}$$

An occurrence of a variable that is not free is bound. We assume that the names of the bound variables are always taken to be different from each other and from the names of the free variables.

This definition differs from the standard one in that it defines that the scope of the parameters Δ in $\Gamma, x[\Delta]{:}A$, $\lambda x[\Delta]{:}A.M$ and $\Pi x[\Delta]{:}A.B$ to be limited to A and that actual parameters (M_i in $x[\langle\Delta_1\rangle M_1 \ldots \langle\Delta_n\rangle M_n]$) and arguments of applications (N in $M \cdot \langle\Delta\rangle N$) are in the scope of extra parameters (Δ_i and Δ resp.). This shows that the application and variable instances can behave as binders.

The process of filling in a value for a parameterized variable is called *instantiation*. As instances of variables provide actual arguments for formal

parameters, we need to propagate the arguments in the term instantiating the variable. This leads to the following definition:

Definition 3.3 [Instantiation] Let Δ be the context $x_1[\Delta_1]{:}A_1, \ldots, x_n[\Delta_n]{:}A_n$ and $m[\Delta] : A$ be a meta-variable. The instantiation of $m[\Delta]$ by an arbitrary term N in the term M (notation $M\{m[\Delta] := N\}$ is defined as follows:

$$\begin{aligned}
s\{m[\Delta] := N\} &= s \\
(m[\langle\Theta_1\rangle M_1 \ldots \langle\Theta_n\rangle M_n])\{m[\Delta] := N\} &= N\{x_1[\Theta_1^*] := M_1^*\} \ldots \{x_n[\Theta_n^*] := M_n^*\} \\
(n[\langle\Theta_1\rangle M_1, \ldots, \langle\Theta_k\rangle M_k])\{m[\Delta] := N\} &= n[\langle\Theta_1^*\rangle M_1^*, \ldots, \langle\Theta_k^*\rangle M_k^*] \\
(M_1 \cdot\langle\,\Theta\rangle M_2)\{m[\Delta] := N\} &= M_1^* \cdot\langle\,\Theta^*\rangle M_2^* \\
(\lambda y[\Theta]{:}U.\, M)\{m[\Delta] := N\} &= \lambda y[\Theta^*]{:}U^*.\, M^* \\
(\Pi y[\Theta]{:}U.\, B)\{m[\Delta] := N\} &= \Pi y[\Theta^*]{:}U^*.\, B^*
\end{aligned}$$

where for readability M^* abbreviates $M\{m[\Delta] := N\}$.

Note that by the $\approx$-convention on variables $\Delta_i \approx \Theta_i$ and this allows us to form the instantiations $\{x_i[\Theta_i{}^*] := u_i{}^*\}$

The well-foundness of instantiation is not completely self-evident, because in the second clause of the definition we apply recursively instantiations to a possibly 'larger' term N. Note however that in that case the contexts involved in the instantiations become strictly 'smaller' (w.r.t the depth of the context, see Definition 3.7) and that ensures the termination of the process.

Example 3.4 A few examples of instantiation:

Term	**Instantiation**	**Result**
$h[]$	$\{h[] := t\}$	$= t$
$h[a]$	$\{h[x{:}A] := x\}$	$= a$
$h[t, \langle x{:}A\rangle p(x,t)]$	$\{h[y{:}A, q[x{:}A]{:}P(x,y)] := q[y]\}$	$= p(t,t)$
$h[g, h[\lambda y{:}A.y, s]]$	$\{h[f{:}\Pi x{:}A.A, x{:}A] := fx\}$	$= g((\lambda y{:}A.y)s)$

Remark 3.5 [Substitution is instantiation with no parameters] Note that if Δ is empty in an instantiation $\{x[\Delta] := t\}$ then the instantiation of x by t in M is exactly the result of the substitution of t for the free occurrences of x in M. For example:

$$(\lambda y[z{:}Ax]{:}B.x)\{x := t\} = \lambda y[z{:}At]{:}B.t$$

Example 3.6 [Variable Capture] Due to the parameters, some variables may get 'captured'. For example in the term

$$(\lambda x{:}A.h[x])\{h[x{:}A] := x\} = \lambda x{:}A.x$$

the variable x is captured by the binder which is in contrast to

$$(\lambda x{:}A.h[\,])\{h[\,] := x\} = \lambda y{:}A.x$$

where x is still free after the instantiation (Note the renaming). In both cases we instantiate h by x but in the first example x is bound and in the second is free. We note that *only variables that have been declared as parameters can get captured.*

The notion of depth reflects the number of levels of parametrization in a context or a parameterized variable.

Definition 3.7 [Depth] The parameter depth of $x[\Delta]$ is by definition $d(\Delta)$, where the depth $d(\Delta)$ of a context Δ is defined as:

$$d(\varepsilon) = 0$$
$$d(\Gamma, x[\Delta]{:}A) = \max(d(\Gamma), d(\Delta) + 1)$$

Example: $d(A{:}*, h[x{:}A]A \to A) = 2$ and $d(A{:}*, x{:}A, h{:}\Pi y{:}A.Bx) = 1$

Proposition 3.8

(i) *If* $\Delta \approx \Theta$ *then* $d(\Delta) = d(\Theta)$.

(ii) *For all* Θ *we have* $\Theta \approx \Theta\{x[\Delta] := N\}$.

Lemma 3.9 *If* $\Delta \approx \Theta$ *and* $\mathsf{dom}(\Delta) = \mathsf{dom}(\Theta)$ *then for all* M *and* N

$$M\{x[\Delta] := N\} = M\{x[\Theta] := N\}$$

Proof. The proof proceeds by induction on the depth of Δ and a nested induction on the structure of M. □

3.2 β-reduction and confluence

Normally β-reduction is defined in terms of the capture-avoiding meta-substitution:

$$(\lambda x{:}A.M)t \to_\beta M[t/x]$$

We extend this definition to our pseudo-terms as follows:

$$(\lambda x[\Theta]{:}A.M) \cdot \langle \Delta \rangle t \to_\beta M\{x[\Delta] := t\} \quad \text{if } \Theta \approx \Delta$$

Note that on unparameterized terms the two reduction relations coincide. The side condition $\Theta \approx \Delta$ is needed, because we need to know that the two contexts have the same structure in order for the instantiation $\{x[\Delta] := t\}$ to be well-defined.

To establish the confluence property we follow the modular confluence proof of Takahashi [17]. Definition 3.10 introduces the notions of parallel

reduction $M \Rightarrow N$ and complete development $\#(M)$ and Lemma 3.11 states the relevant properties:

Definition 3.10 [Parallel reduction $M \Rightarrow N$ and complete development $\#(M)$]

$$M \Rightarrow M \qquad \frac{\Theta_i \Rightarrow \Theta'_i \quad t_i \Rightarrow t'_i}{x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n] \Rightarrow x[\langle\Theta'_1\rangle t'_1 \ldots \langle\Theta'_n\rangle t'_n]}$$

$$\frac{\Delta \Rightarrow \Delta' \quad A \Rightarrow A' \quad M \Rightarrow M'}{\lambda x[\Delta]{:}A.M \Rightarrow \lambda x[\Delta']{:}A'.M'} \qquad \frac{\Delta \Rightarrow \Delta' \quad A \Rightarrow A' \quad B \Rightarrow B'}{\Pi x[\Delta]{:}A.B \Rightarrow \Pi x[\Delta']{:}A'.B'}$$

$$\frac{M \Rightarrow M' \quad N \Rightarrow N' \quad \Delta \Rightarrow \Delta'}{M \cdot\langle\Delta\rangle N \Rightarrow M' \cdot\langle\Delta'\rangle N'} \qquad \frac{M \Rightarrow M' \quad N \Rightarrow N' \quad \Delta \Rightarrow \Delta'}{(\lambda x[\Theta]{:}A].M) \cdot\langle\Delta\rangle N \Rightarrow M'\{x[\Delta'] := N'\}}\,\Theta \approx \Delta$$

$$\begin{aligned}
\#(s) &= s \\
\#(x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]) &= x[\langle\#(\Theta_1)\rangle\#(t_1) \ldots \langle\#(\Theta_n)\rangle\#(t_n)] \\
\#(\lambda x[\Delta]{:}A.M) &= \lambda x[\#(\Delta)]{:}\#(A).\#(M) \\
\#(\Pi x[\Delta]{:}A.B) &= \Pi x[\#(\Delta)]{:}\#(A).\#(B) \\
\#((\lambda x[\Theta]{:}A.M) \cdot\langle\Delta\rangle N) &= \#(M)\{x[\#(\Delta)] := \#(N)\} \\
\#(M \cdot\langle\Delta\rangle N) &= \#(M) \cdot\langle\#(\Delta)\rangle\#(N) \ (M \text{ not an abstraction})
\end{aligned}$$

Lemma 3.11 (Properties of $\Rightarrow$ and $\#$) (i) *If $M_1 \Rightarrow M_2$ and $N_1 \Rightarrow N_2$ then $M_1\{x[\Delta] := N_1\} \Rightarrow M_2\{x[\Delta] := N_2\}$*

(ii) *If $M \Rightarrow N$ then $N \Rightarrow \#(M)$.*

(iii) *If $M \Rightarrow N$ then $M \twoheadrightarrow_\beta N$.*

(iv) *If $M \rightarrow_\beta N$ then $M \Rightarrow N$.*

From (2) it follows easily that $\Rightarrow$ has the diamond property (i.e. if $M \Rightarrow P$ and $M \Rightarrow Q$ then there is a term N such that $P \Rightarrow N$ and $Q \Rightarrow N$). Then, given M, P and Q such that $M \twoheadrightarrow_\beta P$ and $M \twoheadrightarrow_\beta Q$, we have $M \Rightarrow^* P$ and $M \Rightarrow^* Q$ using (4). But then iterating the diamond property for $\Rightarrow$ we get a term N such that $P \Rightarrow^* N$ and $Q \Rightarrow^* N$. But then from (3) we have $P \twoheadrightarrow_\beta N$ and $Q \twoheadrightarrow_\beta N$ using the transitivity of $\twoheadrightarrow_\beta$.

This concludes the proof that β-reduction is confluent.

3.3 Typing system

The derivation rules of a typing system give an inductive definition of the typing relation that assigns types to terms in a given context that specifies the types of the free variables. The standard derivation rules for PTSs (see e.g. [1,6]) are parameterized by three sets $\langle\mathcal{S}, \mathcal{A}, \mathcal{R}\rangle$ and the different PTSs can be obtained by giving particular values to the three parameters. $\mathcal{S}$ is the set of sorts, $\mathcal{A}$ is a subset of $\mathcal{S} \times \mathcal{S}$ and its elements are called axioms. The set

$\mathcal{R}$ is a subset of $\mathcal{S}\times\mathcal{S}\times\mathcal{S}$ and its elements are used to restrict the Π-formation rule.

For the purposes of typing terms with parameterized variables we introduce one extra set $\mathcal{P}$ that would be a subset of $\mathcal{S}\times\mathcal{S}$ and it will be used to denote the dependencies between the type of a parameter of a variable and the type of the variable itself. Hence a PTS with parametric variables will be given by a parametric specification that is a $4-$tuple $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}\rangle$.

Notation 3.12 • *The notion of convertibility between two contexts (notation $\Gamma_1 =_\beta \Gamma_2$) is defined inductively as follows:*
  · $\varepsilon =_\beta \varepsilon$
  · *if* $\Gamma_1 =_\beta \Gamma_2$, $\Delta_1 =_\beta \Delta_2$ *and* $A_1 =_\beta A_2$ *then* $\Gamma_1, x[\Delta_1]{:}A_1 =_\beta \Gamma_2, x[\Delta_2]{:}A_2$

- *We will write* $x[\vec{\Theta}\vec{t}]$ *for* $x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]$.
- *Let* $\Delta = x_1[\Delta_1]{:}A_1, \ldots, x_n[\Delta_n]{:}A_n$ *and* $\vec{t} = \langle t_1, \ldots, t_n\rangle$. *We will write* $\Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta$ *for the conjunction of the judgments* $\Gamma, \Theta_k \vdash t_k{:}\delta_{k-1}A_k$ *with* $k \in [1 \ldots n]$ *where* $\delta_0 = id$, $\delta_{k+1} = \delta_k \circ \{x_{k+1}[\Theta_{k+1}] := t_{k+1}\}$ *and* $\Theta_k =_\beta \delta_{k-1}\Delta_k$.
- *By* $\{\Delta := \vec{\Theta}\vec{t}\}$ *we will denote* δ_n *from above and* $\Delta_{|i}$ *will denote the context*

$$x_1[\Delta_1]{:}A_1, \ldots, x_{i-1}[\Delta_{i-1}]{:}A_{i-1}, \Delta_i$$

Below we give the derivation rules for a PTS with hereditarily parameterized variables. As usual s and s_i denote sorts from $\mathcal{S}$.

Definition 3.13 [Derivation Rules]

$$\frac{}{\vdash s_1{:}s_2} \qquad (s_1, s_2) \in \mathcal{A} \qquad \text{(axiom)}$$

$$\frac{\Gamma, \Delta \vdash A{:}s \quad \Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta}{\Gamma \vdash x[\langle\Theta_1\rangle t_1 \ldots \langle\Theta_n\rangle t_n]{:}A\{\Delta := \vec{\Theta}\vec{t}\}} \qquad x[\Delta]{:}A \in \Gamma \qquad \text{(start)}$$

$$\frac{\Gamma \vdash M{:}B \quad \Gamma, \Delta \vdash A{:}s \quad \Gamma, \Delta_{|i} \vdash A_i{:}s_i}{\Gamma, x[\Delta]{:}A \vdash M{:}B} \qquad (s_i, s) \in \mathcal{P} \qquad \text{(weak)}$$

$$\frac{\Gamma, \Delta \vdash A{:}s_1 \quad \Gamma, x[\Delta]{:}A \vdash B{:}s_2}{\Gamma \vdash \Pi x[\Delta]{:}A.B{:}s_3} \qquad (s_1, s_2, s_3) \in \mathcal{R} \qquad (\Pi)$$

$$\frac{\Gamma, x[\Delta]{:}A \vdash M{:}B \quad \Gamma \vdash \Pi x[\Delta]{:}A.B{:}s}{\Gamma \vdash (\lambda x[\Delta]{:}A.M){:}(\Pi x[\Delta]{:}A.B)} \qquad (\lambda)$$

$$\frac{\Gamma \vdash M{:}\Pi x[\Delta]{:}A.B \quad \Gamma, \Delta \vdash N{:}A}{\Gamma \vdash M \cdot \langle\Delta\rangle N{:}B\{x[\Delta] := N\}} \qquad \text{(app)}$$

$$\frac{\Gamma \vdash M{:}A \quad \Gamma \vdash B{:}s}{\Gamma \vdash M{:}B} \qquad A =_\beta B \qquad \text{(conv)}$$

We briefly comment on the modifications to the rules in order to explain the intuition behind them. In the (start) rule we type the parameterized variables introduced in the context. An instance of a variable is well-typed if it has a correct number and type of arguments. The premise $\Gamma, \Delta \vdash A{:}s$ is necessary in order to ensure that Γ is a valid context in cases when there are no parameters. Each actual parameter t_i can be given a context Θ_i that

locally introduces variables that can be used in t_i. The context Θ_i is required to be β-convertible, but not necessarily equal to $\delta_{i-1}\Delta_i$ because for the Subject Reduction property we should be able to type instances in which β-reductions have been executed in Θ_i.

Using the weakening rule (weak) we can add variables to a context. Note that the parameters of the variable can be used in its type. Very much like in the C^pD^pPTSs [3], by a suitable choice of $\mathcal{P}$ the condition $(s_i, s) \in \mathcal{P}$ is used to restrict the possible parameters that a variable of a given sort can take.

As usual, the Π-formation rule is restricted by $\mathcal{R}$. The new moment is that the bound variable may have parameters. Again, the parameters in Δ can be used in A (but not in B, see Definition 3.2).

The (λ) rule abstracts parameterized variables. If we want to apply such an abstraction to an argument, the argument should be typed in a context that is extended with the parameters. This is done by the (app) rule. Note how the application $\cdot\langle\Delta\rangle$ introduces the parameters in the context of the argument.

We now proceed by establishing the important meta-properties of the system.

Lemma 3.14 (Generation Lemma) *Let $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}\rangle$ be a parametric specification. Then*

(i) *If $\Gamma \vdash s{:}D$ then there is $s' \in \mathcal{S}$ such that $D =_\beta s'$ and $(s, s') \in \mathcal{A}$;*

(ii) *If $\Gamma \vdash x[\vec{\Theta}\vec{t}]{:}D$ then $\Gamma = \Gamma_1, x[\Delta]{:}A, \Gamma_2$ and there is an s such that $\Gamma_1, \Delta \vdash A{:}s$, $\Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta$ and $D =_\beta A\{\Delta := \vec{\Theta}\vec{t}\}$*

(iii) *If $\Gamma \vdash (\Pi x[\Delta]{:}A.B){:}D$ then there are sorts $(s_1, s_2, s_3) \in \mathcal{R}$ such that $\Gamma, \Delta \vdash A{:}s_1$ and $\Gamma, x[\Delta]{:}A \vdash B{:}s_2$ and $D =_\beta s_3$.*

(iv) *If $\Gamma \vdash (\lambda x[\Delta]{:}A.M){:}D$ then there are s and B such that $\Gamma \vdash \Pi x[\Delta]{:}A.B{:}s$, $\Gamma, x[\Delta]{:}A \vdash M{:}B$ and $\Pi x[\Delta]{:}A.B =_\beta D$*

(v) *If $\Gamma \vdash M \cdot\langle\Delta\rangle N{:}D$ then there are A and B such that $\Gamma \vdash M{:}\Pi x[\Delta]{:}A.B$, $\Gamma, \Delta \vdash N{:}B$ and $D =_\beta B\{x[\Delta] := N\}$.*

(vi) *if $\Gamma, x[\Delta]{:}A, \Gamma' \vdash M{:}D$ then there are s and s_i such that $\Gamma, \Delta \vdash A{:}s$ and $\Gamma, \Delta_{|i} \vdash A_i{:}s_i$, $(s_i, s) \in \mathcal{P}$*

Proof. We proceed by induction on the generation of the typing relation $\vdash$. Consider the possible cases for the last rule of a derivation assuming the lemma holds for its subderivations. We treat here only some of the cases:

(start) This means that $D \equiv A\{x[\Delta] := \vec{\Theta}\vec{t}\}$ and

$$\frac{\Gamma, \Delta \vdash A{:}s \quad \Gamma \vdash \vec{\Theta}\vec{t}{:}\Delta}{\Gamma \vdash x[\vec{\Theta}\vec{t}]{:}A\{\Delta := \vec{\Theta}\vec{t}\}}\ x[\Delta]{:}A \in \Gamma$$

From the condition $x[\Delta]{:}A \in \Gamma$ we have $\Gamma = \Gamma_1, x[\Delta]{:}A, \Gamma_2$ and using (6) from the induction hypothesis we have $\Gamma_1, \Delta \vdash A{:}s$.

(weak) Then the last rule looks like this:

$$\frac{\Gamma \vdash M{:}D \quad \Gamma, \Delta \vdash A{:}s \quad \Gamma, \Delta_{|i} \vdash A_i{:}s_i}{\Gamma, x[\Delta]{:}A \vdash M{:}D} \; (s_i, s) \in \mathcal{P}$$

Considering the outermost constructor of M we distinguish five cases and apply the induction hypothesis for $\Gamma \vdash M{:}D$. In this way we prove that the conditions (1) – (5) hold. For (6) we need to use the induction hypothesis and the premises of the rule.

(λ) This means that $D \equiv \Pi x[\Delta]{:}A.B$ and

$$\frac{\Gamma, x[\Delta]{:}A \vdash M{:}B \quad \Gamma \vdash \Pi x[\Delta]{:}A.B{:}s}{\Gamma \vdash (\lambda x[\Delta]{:}A.M){:}(\Pi x[\Delta]{:}A.B)}$$

The statement (6) follows from (6) in the induction hypothesis.

(conv) We use the fact that $=_\beta$ is transitive.

□

Lemma 3.15 (Weakening) *If* $\Gamma_0, \Gamma_1 \vdash M{:}B$, $\Gamma_0, \Delta \vdash A{:}s$ *and* $\Gamma_0, \Delta_{|i} \vdash A_i{:}s_i$ *then* $\Gamma_0, x[\Delta]{:}A, \Gamma_1 \vdash M{:}B$ *where* x *is a fresh variable and* $(s_i, s) \in \mathcal{P}$.

Lemma 3.16 (Substitution Lemma) *If* $\Gamma, x[\Delta]{:}A, \Gamma' \vdash M{:}B$ *and* $\Gamma, \Delta \vdash N{:}A$ *then* $\Gamma, \Gamma'\{x[\Delta] := N\} \vdash M\{x[\Delta] := N\}{:}B\{x[\Delta] := N\}$

Proof. By induction on the depth of Δ and a nested induction on the derivation. □

Lemma 3.17 (Correctness of types) *If* $\Gamma \vdash M{:}A$ *then either* $A =_\beta s$ *or* $\Gamma \vdash A{:}s$ *for some* s.

Proof. By induction on the derivation of $\Gamma \vdash M{:}A$ using Substitution Lemma and Generation Lemma. We treat here only the case of the (app) rule.

By Generation from $\Gamma \vdash M{:}\Pi x[\Delta]{:}A.B$ we get $\Gamma, x[\Delta]{:}A \vdash B{:}s$ for some s. Hence by Substitution $\Gamma \vdash B\{x[\Delta] := N\}{:}s$. □

Lemma 3.18 (Subject Reduction) *Let* $\Gamma \vdash M{:}A$. *Then*

(i) *If* $M \to_\beta N$ *then* $\Gamma \vdash N{:}A$

(ii) *If* $\Gamma \to_\beta \Delta$ *then* $\Delta \vdash M{:}A$

Proof. We will prove the two statements simultaneously by induction on the derivation of $\Gamma \vdash M{:}A$.

- The last rule is (start)

(i) Then $M = x[\vec{\Theta}\vec{t}]$. If the redex is in $\vec{t}$ we apply the induction hypothesis on the respective component $\Gamma, \Theta_k \vdash t_k{:}\delta_{k-1}A_k$. If the redex is in Θ_k we use the induction hypothesis for (2) to get $\Gamma, \Theta'_k \vdash t_k{:}\delta_{k-1}A_k$. Since $\Theta_k \to_\beta \Theta'_k$ and $\Theta_k =_\beta \delta_{k-1}\Delta_k$ we can apply the (start) rule.

- The last rule is (weak)

2. Then $\Gamma = \Gamma', x[\Delta]{:}A$ and the redex can be in Γ', Δ or A. In the first case we simply use the induction hypothesis and apply the (weak) rule to the result. If $\Delta \to_\beta \Delta'$, then by induction $\Gamma, \Delta' \vdash A{:}s$ and we can apply the rule again. If $A \to_\beta A'$, then $\Gamma, \Delta \vdash A{:}s$ and from the hypothesis for (1) we have $\Gamma, \Delta \vdash A'{:}s$.

- The last rule is (app) Let

$$\frac{\Gamma \vdash P{:}\Pi x[\Delta]{:}A.B \quad \Gamma, \Delta \vdash Q{:}A}{\Gamma \vdash P \cdot \langle \Delta \rangle Q{:}B\{x[\Delta] := Q\}}$$

(i) If the redex being contracted is in P, Δ or Q, then we can simply apply the induction hypothesis. If the redex is $P \cdot \langle \Delta \rangle Q$ itself then $P = \lambda y[\Theta]{:}C.R$, reduces to $R\{y[\Delta] := Q\}$. Since $\Gamma \vdash \lambda y[\Theta]{:}C.R{:}\Pi x[\Delta]{:}A.B$ is derivable, then we go up this derivation until the node in which the λ was introduced:

$$\frac{\Gamma', y[\Theta]{:}C \vdash R{:}D \quad \Gamma' \vdash \Pi y[\Theta]{:}C.D{:}s}{\Gamma' \vdash (\lambda y[\Theta]{:}C.R){:}\Pi y[\Theta]{:}C.D}$$

where Γ' is an initial segment of Γ and B is convertible to D. Using weakening we get $\Gamma, y[\Theta]{:}C \vdash R{:}D$ and by Substitution we get $\Gamma \vdash R\{x[\Delta] := Q\}{:}D\{x[\Delta] := Q\}$ which (if necessary using the conversion rule) yields $\Gamma \vdash R\{x[\Delta] := Q\}{:}B\{x[\Delta] := Q\}$

□

Definition 3.19 [Functional specification]A specification $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$ is called *functional* if:

- for all sorts $s_1, s_2,\ s'$ and s'' if $(s_1, s_2, s') \in \mathcal{R}$ and $(s_1, s_2, s'') \in \mathcal{R}$ then $s' = s''$;
- for all sorts $s_1,\ s'$ and s'' if $(s_1, s') \in \mathcal{A}$ and $(s_1, s'') \in \mathcal{A}$ then $s' = s''$.

Lemma 3.20 (Uniqueness of types) *If λS is functional, $\Gamma \vdash M{:}A$ and $\Gamma \vdash M{:}B$ then $A =_\beta B$.*

Proof. By induction on M using Generation. The functionality condition is used when proving the uniqueness of the types of Π-terms and sorts. □

Definition 3.21 [Quasi-Completion] Let $\lambda S = \langle \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$ and $\lambda S' = \langle \mathcal{S}', \mathcal{A}', \mathcal{R}', \mathcal{P}' \rangle$. λS is a quasi-completion of $\lambda S'$ if the following hold:

- $\mathcal{S}' \subseteq \mathcal{S}$, $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{R}' \subseteq \mathcal{R}$;
- For each $s_1, s_2 \in \mathcal{S}'$ there is an $s_3 \in \mathcal{S}$ such that $(s_1, s_2, s_3) \in \mathcal{S}$;

Theorem 3.22 (Strong Normalization) *Let $\lambda S' = \langle \mathcal{S}', \mathcal{A}', \mathcal{R}' \rangle$ be a quasi-completion of $\lambda S^P = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P} \rangle$. Then λS^P is strongly normalizing if $\lambda S'$ is strongly normalizing.*

Proof. We define by induction a reduction- and typing- preserving map $|-|$ from λS^P into $\lambda S'$. Then an assumption that λS^P has an infinite reduction

path induces infinite reduction path in $\lambda S'$ through the map.

$$
\begin{array}{rlrl}
|s| &= s & |\varepsilon| &= \varepsilon \\
|x[\vec{\Theta}\vec{t}]| &= x|^{\lambda}\Theta_1; t_1|\ldots|^{\lambda}\Theta_1; t_1| & |\Gamma, x[\Delta]{:}A| &= |\Gamma|, x{:}|^{\Pi}\Delta; A| \\
|\lambda x[\Delta]{:}A.M| &= \lambda x{:}|^{\Pi}\Delta; A|.|M| & & \\
|\Pi x[\Delta]{:}A.M| &= \Pi x{:}|^{\Pi}\Delta; A|.|M| & |^{\sigma}\Gamma, x[\Delta]{:}A; M| &= |^{\sigma}\Gamma; \sigma x{:}|^{\Pi}\Delta; A|.M| \\
|P\cdot\langle\Delta\rangle Q| &= |P||^{\lambda}\Delta; Q| & |^{\sigma}\varepsilon; M| &= M
\end{array}
$$

Then we have

(i) If $\Gamma \vdash_{\lambda S^P} M{:}A$ then $|\Gamma| \vdash_{\lambda S'} |M|{:}|A|$;

(ii) If $M \to_\beta N$ then $|M| \to_\beta^+ |N|$.

For (1), since $\lambda S'$ is a completion of λS^P, we can form the types generated by $|^{\Pi}-;-|$ and those types can be used to type the terms generated by $|^{\lambda}-;-|$. The typability of the rest of the terms is not problematic. As for (2), from the definition of $|-|$ it is clear that a redex is mapped into a redex. However, each occurrence of a paramaterized variable after propagating the instantiation generates as many redexes as the number of its paramaters. After we reduce those, we are done. □

Corollary 3.23 *The systems of the λ-cube extended with hereditary parameters are strongly normalizing.*

Proof. The Extended Calculus of Constructions (ECC) of Luo [9] is a quasi-completion of all the systems of the λ-cube with hereditarily parameterized variables. Since ECC is strongly normalizing, by Theorem 3.22 each of the systems of the cube is strongly normalizing. □

4 Future Work

After obtaining a calculus that can express both open terms and the basic operations on them explicitly we intend to investigate the possibilities of extending it with operations that could make it applicable for modelling real tactics, not only tactic instances. To do that we need to internalize other essential operations like unification and recursion. Ultimately we would like to be able to have tactic terms like the one below that represents the propsitional tactic Apply:

$$
\begin{array}{l}
\mathsf{Apply}[\varphi, \psi : \mathsf{Prop}, thm : \psi] : \varphi := \\
\quad (\varphi \sim \psi).thm \mid \\
\quad ?A, B : \mathsf{Prop}.(\psi \sim A \to B).?m : A.\mathsf{Apply}[\varphi, B, (thm\ m)]
\end{array}
$$

When given two propositions φ and ψ and a proof of ψ this tactic tries to unify φ and ψ and if this is successful it returns a proof of ψ that in this case

is also a proof of φ. If the unification fails, it checks whether ψ is an arrow type by trying to unify it with $A \to B$ where A and B are meta-variables freshly introduced by the binder $?A, B : \mathsf{Prop}$. If this is the case, the tactic makes a recursive call by eliminating the argument A with a freshly introduced meta-variable m.

Designing a calculus capable of representing tactics like Apply is a major challenge, but we hope that the present paper is the right first step towards this goal.

References

[1] Henk Barendregt. Lambda calculi with types. In Abramsky et al., editor, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.

[2] S. Berardi. Towards a methematical analysis of the Coquand-Huet calculus and the other systems in Barendregt's cube. Technical report, Carnegie-Mellon University and Universita di Torino, 1989.

[3] R. Bloo, Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. Parameters in Pure Type Systems. In *Proceedings of LATIN'02*. Springer, 2002.

[4] Mirna Bognar. *Contexts in Lambda Calculus*. PhD thesis, Vrije Universiteit Amsterdam, 2002.

[5] N.G. de Bruijn. A Survey of the Project AUTOMATH. In Hindley and Seldin, editors, *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[6] Herman Geuvers. *Logics and Type systems*. PhD thesis, University of Nijmegen, 1993.

[7] Herman Geuvers and G.I. Jojgov. Open Proofs and Open Terms: a Basis for Interactive Logic. In Bradfield, editor, *Proceedings of CSL'02*. Springer, 2002.

[8] Gueorgui Jojgov. Holes with Binding Power. In *Proceedings of TYPES'02*. Springer, 2003.

[9] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, July 1990.

[10] Zhaohui Luo. PAL$^+$: A lambda-free logical framework. *Journal of Functional Programming*, to appear.

[11] Lena Magnusson. *The Implementation of ALF - a Proof Editor based on Martin-Löf Monomorphic Type Theory with Explicit Substitutions*. PhD thesis, Chalmers University of Technology / Göteborg University, 1995.

[12] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999.

[13] Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 1992.

[14] César A. Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory.* PhD thesis, INRIA, November 1997.

[15] P. Severi and E. Poll. Pure Type Systems with definitions. In *Proc. of LFCS'94, St. Petersburg, Russia*, number 813 in LNCS, Berlin, 1994. Springer Verlag.

[16] M. Strecker. *Construction and Deduction in Type Theories.* PhD thesis, Universität Ulm, 1999.

[17] M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118, 1995.

[18] J. Terlouw. Een nadere bewijstheoretische analyse van GSTTs. Technical report, University of Nijmegen, 1989.